Inteligência Computacional

PROJETO – FASE2

Elementos do grupo:
Luís Henrique P. O. Travassos, nº2021136600
Rodrigo Ramalho Ferreira, nº2021139149

isec Engenharia

Computação Swarm (PSO)

- **Definição:**
  - Inspirada em enxames biológicos(formigas, abelhas).
  - Colaboração descentralizada de agentes autônomos.
- **Aplicação na IA:**
  - Treino eficiente de redes neuronais artificias
  - Otimização de hiperparâmetros e treino distribuído.
  - Destaque na deteção de anomalias em grandes conjuntos de dados.
- **Sinergia com IA:**
  - Reflete a tendência de imitar processos naturais na computação.
  - Adaptação e eficiência em diversas áreas, desde análise de dados até robótica.

Computação Swarm (GWO)

- **Influência Biológica:**
  - Comportamento social e de caça de lobos cizentos
  - Hierarquia: alfa, beta, delta, ômegas.
- **Funcionamento:**
  - População de lobos busca a solução ótima.
  - Hierarquia influencia dinâmica de busca.
  - Atualização das posições guiada pelos líderes.
- **Vantagens:**
  - Hierarquia evita convergência prematura.
  - Menos parâmetros a ajustar comparado ao PSO.
  - Eficaz em espaços de busca complexos.
- **Desvantagens:**
  - Maior intensidade computacional devido à complexidade social.
  - Hierarquia pode limitar diversidade de soluções.
  - Desafios em sair de mínimos locais.

Análise da Influência do PSO na função de Ackley:

Demonstração do Código

- Implementação do PSO para encontrar o mínimo global na função Ackley.
- Objetivo de minimizar o “custo” calculado pela função, que pode ser de duas ou três dimensões.
- Parâmetros como confiança pessoal (c1), confiança social (c2), e inércia (w) são ajustados para explorar e explorar o espaço de busca.
- Inicialmente, 30 partículas em 100 iterações.
- Resultado eficaz: mínimo global em (0,0) ou (0,0,0).

```python
import numpy as np
import pyswarms as ps
```

```python
# Definição da função de Ackley
def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = len(x)  # Dimensão do vetor de entrada
    sum_sq_term = -a * np.exp(-b * np.sqrt(np.sum(x**2) / d))
    cos_term = -np.exp(np.sum(np.cos(c * x) / d))
    result = sum_sq_term + cos_term + a + np.exp(1)
    return result
```

```python
# Definindo os limites do espaço de busca para a função de Ackley
# Estes são os limites comuns para testar a função de Ackley
bounds = (np.array([-32.768] * 2), np.array([32.768] * 2))  # Ajuste para 2 dimensões

# Definindo os limites do espaço de busca para a função de Ackley para 3 dimensões
#bounds = (np.array([-32.768] * 3), np.array([32.768] * 3))  # Ajuste para 3 dimensões

# Configurações do PSO
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

# Inicializando o PSO
optimizer = ps.single.GlobalBestPSO(n_particles=30,
                                    dimensions=2,
                                    options=options,
                                    bounds=bounds)
```

```python
# Executando a otimização
cost, pos = optimizer.optimize(ackley_function, iters=100)

# Imprimindo o resultado da otimização
print(f"Melhor custo encontrado: {cost}")
print(f"Melhor posição encontrada: {pos}")
```

Análise de resultados:

Análise dos diferentes parâmetros

Parâmetros	Melhor Custo (2 Dim.)	Melhor Posição (2 Dim.)	Melhor Custo (3 Dim.)	Melhor Posição (3 Dim.)	Objetivo
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 30** **iters = 100**	21.13	(-23.51, 24.94)	20.96	(-26.13, 21.90, 25.31)	Valores iniciais e escolhidos aleatoriamente, só para servir de início à análise.
c1 = 1.0 **c2 = 1.0** **w = 0.9** **n_particles = 30** **iters = 100**	17.49	(4.58, -2.85)	20.69	(4.79, -20.04, 17.33)	Avaliar o impacto de um maior equilíbrio entre a componente cognitiva e a componente social no desempenho do algoritmo.
c1 = 0.5 **c2 = 0.3** **w = 0.4** **n_particles = 30** **iters = 100**	21.00	(-26.95, -6.71)	20.96	(-6.49, 19.90, 17.51)	Observar como uma menor inércia (que promove uma maior exploração do espaço de busca) afeta a capacidade do algoritmo de encontrar o mínimo global.

Análise de resultados:

Análise dos diferentes parâmetros

Parâmetros	Melhor Custo (2 Dim.)	Melhor Posição (2 Dim.)	Melhor Custo (3 Dim.)	Melhor Posição (3 Dim.)	Objetivo
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 100** **iters = 100**	20.26	(-15.89, -3.66)	17.76	(-23.32, 4.08, 2.01)	Determinar se um enxame maior melhora a convergência para o mínimo global devido a uma maior diversidade de soluções sendo exploradas.
c1 = 0.5 **c2 = 0.3** **w = 0.9** **n_particles = 30** **iters = 50**	21.22	(-7.25, 30.44)	20.96	(10.30, 20.47, -0.92)	Verificar se o algoritmo ainda consegue encontrar uma boa solução com menos iterações.
c1 = 1.0 **c2 = 1.0** **w = 0.9** **n_particles = 100** **iters = 100**	21.23	(-17.02, -28.88)	21.39	(-22.39, -30.75, -0.51)	Com a análise dos resultados anteriores iremos, como análise final, aumentar os valores de c1, c2 e n_particles, enquanto mantemos o valor de w e iters.

Análise de resultados:

Análise final

- Ajustes em c1 e c2, maior número de partículas, e iterações mostraram melhorias.
- Mínimo global não atingido, o que sugere um espaço para aprimoramentos nos parâmetros ou metodologia.
- Mantendo o valor de w e iters constantes, a análise final destaca um equilíbrio entre exploração do espaço de busca e convergência rápida. Ao realçar a complexidade do PSO e a importância de ajustes precisos para otimização eficaz em vários contextos, enfatiza-se a necessidade de uma abordagem adaptativa e personalizada.

Análise da Influência do GWO na função de Ackley (melhores parâmetros):

Demonstração do Código

- Implementação do GWO com os melhores parâmetros do PSO para encontrar o mínimo global na função Ackley.
- Objetivo de minimizar o “custo” calculado pela função, que pode ser de duas ou três dimensões.
- Sem parâmetros, ao invés do PSO.
- 100 partículas em 100 iterações.
- Resultado eficaz: mínimo global em (0,0) ou (0,0,0).

```python
from SwarmPackagePy import gwo
import numpy as np
```

```python
# Definir a função objetivo, por exemplo, a função Ackley
def ackley_function(x):
    a = 20
    b = 0.2
    c = 2 * np.pi
    d = len(x)  # Dimensão do vetor de entrada
    sum_sq_term = -a * np.exp(-b * np.sqrt(sum(x**2) / d))
    cos_term = -np.exp(sum(np.cos(c * x) / d))
    result = a + np.exp(1) + sum_sq_term + cos_term
    return result
```

```python
# Parâmetros do GWO
n_agents = 100
n_iterations = 100
dimension = 3  # Duas dimensões
lower_bound = -32.768
upper_bound = 32.768

# Executar o algoritmo GWO
optimizer = gwo(n_agents, ackley_function, lower_bound, upper_bound, dimension, n_iterations)
```

```python
best_position = optimizer.get_Gbest()

# Exibir os resultados
print(f"Melhor posição: {best_position}")
```

```
Melhor posição: [2.9278990147543366e-16, 1.1782291597397856e-17, 2.828315148236302e-16]
```

Análise de resultados do GWO:

Análise com os melhores parâmetros do PSO em GWO:

Parâmetros	Melhor Posição (2 Dim.)	Melhor Posição (3 Dim.)	Objetivo
n_agents = 100 **n_iterations = 100**	(-3.90, -1.48)	(2.93, 1.18, 2.83)	Comparar os resultados obtidos pelo algoritmo PSO com o algoritmo GWO

Conclusões:

Com o uso dos “melhores” valores para os parâmetros do algoritmo PSO no algoritmo GWO podemos chegar à conclusão que apesar do GWO possuir menos parâmetros de configuração ainda assim aproxima-se muito mais do resultado pretendido, neste caso o mínimo global (0, 0) ou (0, 0, 0), ao contrário do algoritmo PSO, podendo assim afirmar-se, necessitando de mais experimentos, de que este algoritmo têm um melhor desempenho.

Comparação entre GWO e PSO na Otimização da Arquitetura:

Análise do cenário de testes

- Desafios na aplicação de GWO e PSO devido à pesada carga computacional.
- Restrições de tempo limitam a exploração extensiva dos hiperparâmetros.
- Apenas dois hiperparâmetros ajustáveis: learning-rate e dropout-rate.
- Implementação com 5 partículas e 5 iterações limita a avaliação da performance dos algoritmos.
- Dificuldade em obter conclusões robustas devido à amostragem limitada.
- Necessidade de testes, mais extensivos e ajustes finos de hiperparâmetros.

Comparação entre GWO e PSO na Otimização da Arquitetura:

Apresentação código PSO (Particle Swarm Optimization)

- Uso do PSO para treinar uma CNN otimizando taxa de aprendizagem e taxa de desativação.
- Função de treino define e compila a CNN, usando validação para avaliação.
- Função de fitness avalia a adequação dos hiperparâmetros.
- PSO encontra melhores hiperparâmetros em 5 iterações.

```
# Defina as funções de treinamento de rede e fitness para o PSO
def train_network(hyperparameters, train_x, train_y, val_x, val_y):

    learning_rate, dropout_rate = hyperparameters

    # Definindo o modelo da rede neural convolucional
    model = Sequential([
        Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
        MaxPooling2D((2, 2)),

        Flatten(),

        Dropout(dropout_rate),

        Dense(len(street_types), activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(train_x,
              train_y,
              epochs=5,
              validation_data=(val_x, val_y),
              verbose=0)

    validation_loss, validation_accuracy = model.evaluate(val_x, val_y, verbose=0)

    return validation_loss
```

```
def fitness_function(x, train_x, train_y, val_x, val_y):

    n_particles = x.shape[0]

    losses = []

    for i in range(n_particles):
        # x[i] contém os hiperparâmetros para a i-ésima partícula
        hyperparameters = x[i]
        loss = train_network(hyperparameters, train_x, train_y, val_x, val_y)
        losses.append(loss)

    return np.array(losses)
```

```
# Inicializando o otimizador GlobalBestPSO do pyswarms
bounds = [(0.0001, 0.1), (0.0, 0.5)]  # (min_learning_rate, max_learning_rate), (min_dropout_rate, max_dropout_rate)
options = {'c1': 0.5, 'c2': 0.3, 'w': 0.9}

optimizer = ps.single.GlobalBestPSO(n_particles=5,
                                    dimensions=2,
                                    options=options,
                                    bounds=bounds)

# Executando o PSO para encontrar os melhores hiperparâmetros
cost, best_pos = optimizer.optimize(fitness_function,
                                    iters=5,
                                    train_x=train_x,
                                    train_y=train_y,
                                    val_x=val_x,
                                    val_y=val_y)

best_learning_rate, best_dropout_rate = best_pos
```

```
2023-11-08 11:44:11,151 - pyswarms.single.global_best - INFO - Optimize for 5 iters with {'c1': 0.5, 'c2': 0.3, 'w': 0.9}
pyswarms.single.global_best: 100%|██████████|5/5, best_cost=0.196
2023-11-08 16:33:36,606 - pyswarms.single.global_best - INFO - Optimization finished | best cost: 0.19598796963691711, bes
t pos: [0.00018259 0.10997558]
```

```
final_model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
    MaxPooling2D((2, 2)),

    Flatten(),

    Dropout(best_dropout_rate),

    Dense(len(street_types), activation='softmax')
])

final_model.compile(optimizer=Adam(best_learning_rate),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

history = final_model.fit(train_x,
                          train_y,
                          epochs=5,
                          validation_data=(val_x, val_y))
```

Comparação entre GWO e PSO na Otimização da Arquitetura:

Análise de resultados PSO (Particle Swarm Optimization)

- Melhores hiperparâmetros: taxa de aprendizagem ~0.00018, taxa de desativação(dropout) ~0.11.
- Precisão do modelo: 92.50%.
- Desempenho equilibrado e robusto em todas as classes.

```
Best learning rate: 0.00018258859132538424
Best dropout rate: 0.10997558109383135

Test accuracy: 92.50%

Confusion Matrix:
[[176  20   4]
 [  8 192   0]
 [ 12   1 187]]

Classification Report:
              precision    recall  f1-score   support

       clean       0.90      0.88      0.89       200
      litter       0.90      0.96      0.93       200
     recycle       0.98      0.94      0.96       200

    accuracy                           0.93       600
   macro avg       0.93      0.92      0.93       600
weighted avg       0.93      0.93      0.93       600



AUC for class clean: 0.98
AUC for class litter: 0.99
AUC for class recycle: 0.99
```

*Notas sobre valores:

- **Best Learning Rate:** Este valor representa a taxa na qual o modelo de aprendizagem de máquina atualiza os seus pesos durante o treino.
- **Best Dropout Rate:** Indica a percentagem de neurónios que são aleatoriamente ignorados durante o treino, para prevenir o sobreajuste (overfitting).
- **Test Accuracy:** Reflete a percentagem de previsões corretas feitas pelo modelo em relação ao total de previsões.
- **Confusion Matrix:** Mostra o número de previsões corretas e incorretas divididas por classe.
- Classification Report:
  - Precision: A precisão indica a proporção de identificações positivas corretas em relação ao total de identificações positivas.
  - Recall: O recall mede a proporção de positivos reais corretamente identificados pelo modelo.
  - F1-Score: O F1-score é a média harmônica entre precisão e recall, oferecendo um balanço entre ambos.
  - Support: Indica o número real de ocorrências de cada classe nos dados.
- **AUC (Area Under Curve):** Valores indicam a capacidade do modelo de distinguir entre as classes. Valores mais próximos de 1.0 mostram uma melhor distinção.

Comparação entre GWO e PSO na Otimização da Arquitetura:

Apresentação do código GWO (Gray Wolf Optimization)

- GWO ajusta hiperparâmetros para treinar uma CNN.
- Funções para treino e fitness.
- GWO encontra melhores hiperparâmetros em iterações.

```python
def train_network(hyperparameters, train_x, train_y, val_x, val_y):
    learning_rate, dropout_rate = hyperparameters
    im_size = train_x.shape[1]

    # Definindo o modelo da rede neural
    model = Sequential([
        Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
        MaxPooling2D((2, 2)),
        Flatten(),
        Dropout(dropout_rate),  # Usar a variável desempacotada
        Dense(len(street_types), activation='softmax')
    ])

    model.compile(optimizer=Adam(learning_rate=learning_rate),
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])

    model.fit(train_x,
              train_y,
              epochs=5,
              validation_data=(val_x, val_y),
              verbose=0)

    validation_loss, validation_accuracy = model.evaluate(val_x, val_y, verbose=0)
    return validation_loss
```

```python
# Defina a função de fitness que será usada pelo GWO
def fitness_function(hyperparameters):
    loss = train_network(hyperparameters, train_x, train_y, val_x, val_y)
    print(f"Particle: {hyperparameters}, Loss: {loss}")
    return loss
```

```python
# Parâmetros do GWO
n_agents = 5
n_iterations = 5
dimensions = 2
lower_bound = [0.0001, 0.0]  # Limites inferiores para taxa de aprendizado e dropo
upper_bound = [0.1, 0.5]     # Limites superiores para taxa de aprendizado e dropo

# Inicializar e executar o otimizador GWO
optimizer = gwo(n_agents,
                fitness_function,
                lower_bound,
                upper_bound,
                dimensions,
                n_iterations)

# Desempacotando os melhores hiperparâmetros encontrados
best_learning_rate, best_dropout_rate = optimizer.get_Gbest()
```

```python
final_model = Sequential([
    Conv2D(64, (3, 3), activation='relu', input_shape=(im_size, im_size, 3)),
    MaxPooling2D((2, 2)),

    Flatten(),

    Dropout(best_dropout_rate),

    Dense(len(street_types), activation='softmax')
])

final_model.compile(optimizer=Adam(best_learning_rate),
                    loss='sparse_categorical_crossentropy',
                    metrics=['accuracy'])

history = final_model.fit(train_x,
                          train_y,
                          epochs=5,
                          validation_data=(val_x, val_y))
```

Comparação entre GWO e PSO na Otimização da Arquitetura:

Análise de resultados GWO (Gray Wolf Optimization)

- Melhores hiperparâmetros: taxa de aprendizagem ~0.00013, taxa de desativação(dropout) ~0.10.
- Precisão do modelo: 91.67%.
- Desempenho equilibrado e robusto em todas as classes.

```
Best learning rate: 0.00012542080373561626
Best dropout rate: 0.10186120443028664

Test accuracy: 91.67%

Confusion Matrix:
[[176   8  16]
 [ 14 184   2]
 [  9   1 190]]

Classification Report:
              precision    recall  f1-score   support

       clean       0.88      0.88      0.88       200
      litter       0.95      0.92      0.94       200
     recycle       0.91      0.95      0.93       200

    accuracy                           0.92       600
   macro avg       0.92      0.92      0.92       600
weighted avg       0.92      0.92      0.92       600



AUC for class clean: 0.97
AUC for class litter: 0.98
AUC for class recycle: 0.99
```

Conclusão e discussão de resultados

- GWO: 91.67% de precisão, PSO: 92.50% de precisão.
- Desempenho equilibrado para todas as classes.
- Diferenças nas melhores precisões por classe.
- AUC excepcionalmente alta em ambas as otimizações.
- GWO e PSO são confiáveis para otimizar hiperparâmetros.
- Escolha entre os algoritmos pode depender de preferências ou restrições específicas.
- Modelos equilibrados e robustos, validando eficácia em problemas complexos.

Referências

- SwarmPackagesPy:
  - https://github.com/SISDevelop/SwarmPackagePy
- ComputaçãoSwarm:
  - https://pt.wikipedia.org/wiki/Intelig%C3%AAncia_de_enxame
- Ackley’s Function :
  - https://en.wikipedia.org/wiki/Ackley_function